Immutability

Lecture Question

Restriction: No state is allowed in this question. Specifically, the keyword "var" is banned

Question: In a package named "functions" write a class named Point with the following features:

- Has a constructor that takes 2 values (Use val) of type Double named "x" and "y"
- A method named "add" that takes a Point and returns a Point that is the component-wise addition of this Point and the input Point
  - Ex. (1.0, 2.0) + (4.0, 1.0) = (5.0, 3.0)
- A method named "multiplyByScalar" that takes a Double and returns a new Point that is this Point multiplied by the input
  - Ex. 5.0 * (1.0, 2.0) = (5.0, 10.0)

Testing: In a package named "tests" create a class named "TestPoint" as a test suite that tests all the functionality listed above

Immutable Objects

- Values stored in state variables cannot change
- Immutable objects are stored on the heap just like any other object
  - But we don't worry about the state changing when we pass the reference to a method/function

- What if an immutable object needs to change state?
  - Create a copy of the object with the change applied

Immutable Objects

- This ImmutableCounter class takes and initial value in its constructor and has method to increment and decrement this value
- The internal Int is a value and cannot change
  - It also can't be accessed (Artificial restriction to show more recursion)

```
class ImmutableCounter(counter: Int) {

  def printCount():Unit = {
    println(this.counter)
  }

  def increase(): ImmutableCounter = {
    new ImmutableCounter(this.counter + 1)
  }

  def decrease(): ImmutableCounter = {
    new ImmutableCounter(this.counter - 1)
  }

}
```

```
def updateCounter(n: Int, counter: ImmutableCounter): ImmutableCounter = {
  if(n==0){
    counter
  }else if(n < 0){
    updateCounter(n+1, counter.decrease())
  }else{
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: ImmutableCounter = new ImmutableCounter(10)
  val counter2: ImmutableCounter = updateCounter(20, counter)

  counter.printCount()
  counter2.printCount()
}
```

Immutable Objects

- Since the Int cannot change
  - We simulate changes by creating a new object on the heap with the change applied
- Create and return a new ImmutableCounter whenever a "change" is made

```
class ImmutableCounter(counter: Int) {

 def printCount():Unit = {
    println(this.counter)
  }

  def increase(): ImmutableCounter = {
    new ImmutableCounter(this.counter + 1)
  }

  def decrease(): ImmutableCounter = {
    new ImmutableCounter(this.counter - 1)
  }

}
```

```
def updateCounter(n: Int, counter: ImmutableCounter): ImmutableCounter = {
  if(n==0){
    counter
  }else if(n < 0){
    updateCounter(n+1, counter.decrease())
  }else{
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: ImmutableCounter = new ImmutableCounter(10)
  val counter2: ImmutableCounter = updateCounter(20, counter)

  counter.printCount()
  counter2.printCount()
}
```

Immutable Objects

- Since we return a new ImmutableCounter
  - We must use this return value or we will not see the change

```
class ImmutableCounter(counter: Int) {

 def printCount():Unit = {
    println(this.counter)
  }

  def increase(): ImmutableCounter = {
    new ImmutableCounter(this.counter + 1)
  }

  def decrease(): ImmutableCounter = {
    new ImmutableCounter(this.counter - 1)
  }

}
```

```
def updateCounter(n: Int, counter: ImmutableCounter): ImmutableCounter = {
  if(n==0){
    counter
  }else if(n < 0){
    updateCounter(n+1, counter.decrease())
  }else{
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: ImmutableCounter = new ImmutableCounter(10)
  val counter2: ImmutableCounter = updateCounter(20, counter)

  counter.printCount()
  counter2.printCount()
}
```

Immutable Objects

- What if we want to increment this object 10 times?
- Since we [artificially] restrict access to the Int we can only increment and decrement
- We could use a loop and reassign a variable at each iteration (requires var)

```
class ImmutableCounter(counter: Int) {

 def printCount():Unit = {
    println(this.counter)
  }

  def increase(): ImmutableCounter = {
    new ImmutableCounter(this.counter + 1)
  }

  def decrease(): ImmutableCounter = {
    new ImmutableCounter(this.counter - 1)
  }

}
```

```
def updateCounter(n: Int, counter: ImmutableCounter): ImmutableCounter = {
  if(n==0){
    counter
  }else if(n < 0){
    updateCounter(n+1, counter.decrease())
  }else{
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: ImmutableCounter = new ImmutableCounter(10)
  val counter2: ImmutableCounter = updateCounter(20, counter)

  counter.printCount()
  counter2.printCount()
}
```

Immutable Objects

- What if we want to increment this object 10 times?
- Use a recursive approach
  - Base case of n==0
  - Recursively increment/decrement and make a recursive call with n closer to 0

```
class ImmutableCounter(counter: Int) {

 def printCount():Unit = {
    println(this.counter)
  }

  def increase(): ImmutableCounter = {
    new ImmutableCounter(this.counter + 1)
  }

  def decrease(): ImmutableCounter = {
    new ImmutableCounter(this.counter - 1)
  }

}
```

```
def updateCounter(n: Int, counter: ImmutableCounter): ImmutableCounter = {
  if(n==0){
    counter
  }else if(n < 0){
    updateCounter(n+1, counter.decrease())
  }else{
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: ImmutableCounter = new ImmutableCounter(10)
  val counter2: ImmutableCounter = updateCounter(20, counter)

  counter.printCount()
  counter2.printCount()
}
```

Strings are Immutable

Stack

RAM	
Main Frame	args
	name:course, value:@1
nerf Frame	name:input, value:@1

Heap

RAM @1
Object of type String
"CSE116"

```
def nerf(input: String): Unit = {
  input.replace("6", "5")
}

def amplify(input: String): String = {
  input.replace("116", "250")
}

def main(args: Array[String]): Unit = {

  val course: String = "CSE116"
  nerf(course)
  val dataStructures: String = amplify(course)

  course + " is great!"
  val courseString = course + " is fun!"

  println(course)
  println(dataStructures)
  println(courseString)
}
```

- The main method creates a new String on the stack and passes a reference to it to the nerf method
- We would usually expect to see changes made to this object by the method

Stack

RAM	
Main Frame	args
	name:course, value:@1
nerf Frame	name:input, value:@1
	replace method resolves to @2

Heap

RAM @1
Object of type String
"CSE116"

RAM @2
Object of type String
"CSE115"

```
def nerf(input: String): Unit = {
  input.replace("6", "5")
}

def amplify(input: String): String = {
  input.replace("116", "250")
}

def main(args: Array[String]): Unit = {

  val course: String = "CSE116"
  nerf(course)
  val dataStructures: String = amplify(course)

  course + " is great!"
  val courseString = course + " is fun!"

  println(course)
  println(dataStructures)
  println(courseString)
}
```

- The method "replaces" all instance of the substring "6" with "5"
- The "change" is made by creating a new String

Stack

RAM	
Main Frame	args
	name:course, value:@1
nerf Frame	returns Unit

Heap

RAM @1
Object of type String
"CSE116"

RAM @2
Object of type String
"CSE115"

```
def nerf(input: String): Unit = {
  input.replace("6", "5")
}

def amplify(input: String): String = {
  input.replace("116", "250")
}

def main(args: Array[String]): Unit = {

  val course: String = "CSE116"
  nerf(course)
  val dataStructures: String = amplify(course)

  course + " is great!"
  val courseString = course + " is fun!"

  println(course)
  println(dataStructures)
  println(courseString)
}
```

- Since this method has a return type of Unit, the reference @2 is not returned
- The String @2 is still on the heap

Stack

RAM	
Main Frame	args
	name:course, value:@1

Heap

RAM @1
Object of type String
"CSE116"

RAM @2
Object of type String
"CSE115"

```
def nerf(input: String): Unit = {
  input.replace("6", "5")
}

def amplify(input: String): String = {
  input.replace("116", "250")
}

def main(args: Array[String]): Unit = {

  val course: String = "CSE116"
  nerf(course)
  val dataStructures: String = amplify(course)

  course + " is great!"
  val courseString = course + " is fun!"

  println(course)
  println(dataStructures)
  println(courseString)
}
```

- After the call to nerf resolves
  - The stack is in the same state as it was before the method call
- There is an extra String on the Stack
  - [It can be garbage collected]

Stack

RAM	
Main Frame	args
	name:course, value:@1
amplify Frame	name:input, value:@1

Heap

RAM @1
Object of type String
"CSE116"

RAM @2
Object of type String
"CSE115"

RAM @3
Object of type String
"CSE250"

```
def nerf(input: String): Unit = {
  input.replace("6", "5")
}

def amplify(input: String): String = {
  input.replace("116", "250")
}

def main(args: Array[String]): Unit = {

  val course: String = "CSE116"
  nerf(course)
  val dataStructures: String = amplify(course)

  course + " is great!"
  val courseString = course + " is fun!"

  println(course)
  println(dataStructures)
  println(courseString)
}
```

- The next method call also creates a new String on the heap
- Replaces "116" with "250"

Stack

RAM	
Main Frame	args
	name:course, value:@1
amplify Frame	returns the reference @3

Heap

RAM @1
Object of type String
"CSE116"

RAM @2
Object of type String
"CSE115"

RAM @3
Object of type String
"CSE250"

```
def nerf(input: String): Unit = {
  input.replace("6", "5")
}

def amplify(input: String): String = {
  input.replace("116", "250")
}

def main(args: Array[String]): Unit = {

  val course: String = "CSE116"
  nerf(course)
  val dataStructures: String = amplify(course)

  course + " is great!"
  val courseString = course + " is fun!"

  println(course)
  println(dataStructures)
  println(courseString)
}
```

- Method returns a reference to the new String that was created

Stack

RAM	
Main Frame	args
	name:course, value:@1
	name:dataStructures, value:@3

Heap

RAM @1
Object of type String
"CSE116"

RAM @2
Object of type String
"CSE115"

RAM @3
Object of type String
"CSE250"

```
def nerf(input: String): Unit = {
  input.replace("6", "5")
}

def amplify(input: String): String = {
  input.replace("116", "250")
}

def main(args: Array[String]): Unit = {

  val course: String = "CSE116"
  nerf(course)
  val dataStructures: String = amplify(course)

  course + " is great!"
  val courseString = course + " is fun!"

  println(course)
  println(dataStructures)
  println(courseString)
}
```

- The reference is stored in a variable in the main method

Stack

RAM	
Main Frame	args
	name:course, value:@1
	name:dataStructures, value:@3

Heap

RAM @1
Object of type String
"CSE116"

RAM @2
Object of type String
"CSE115"

RAM @3
Object of type String
"CSE250"

RAM @4
Object of type String
"CSE116 is great!"

```scala
def nerf(input: String): Unit = {
  input.replace("6", "5")
}

def amplify(input: String): String = {
  input.replace("116", "250")
}

def main(args: Array[String]): Unit = {

  val course: String = "CSE116"
  nerf(course)
  val dataStructures: String = amplify(course)

  course + " is great!"
  val courseString = course + " is fun!"

  println(course)
  println(dataStructures)
  println(courseString)
}
```

- We create another new String in main
- The reference is never stored in a variable
- Never see this String in our code

Stack

RAM	
Main Frame	args
	name:course, value:@1
	name:dataStructures, value:@3
	name:courseString, value:@5

```
def nerf(input: String): Unit = {
  input.replace("6", "5")
}

def amplify(input: String): String = {
  input.replace("116", "250")
}

def main(args: Array[String]): Unit = {

  val course: String = "CSE116"
  nerf(course)
  val dataStructures: String = amplify(course)

  course + " is great!"
  val courseString = course + " is fun!"

  println(course)
  println(dataStructures)
  println(courseString)
}
```

Heap

RAM @1
Object of type String
"CSE116"

RAM @2
Object of type String
"CSE115"

RAM @3
Object of type String
"CSE250"

RAM @4
Object of type String
"CSE116 is great!"

RAM @5
Object of type String
"CSE116 is fun!"

- Another new String is created and stored in a value

Lists are Immutable

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
firstNPrimes Frame	name:n, value:2
firstNPrimes Frame	name:n, value:1

Heap

RAM @1
Object of type List[Int]
2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- Recursive calls are added to the stack until we reach the base case of n == 1
- Create a new immutable List on the heap

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
firstNPrimes Frame	name:n, value:2
firstNPrimes Frame	returns the reference @1

Heap

RAM @1
Object of type List[Int]
2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- The base case returns a reference to the List it created
- This List is immutable so it will never change
  - Even though it's reference is passed around different frames

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
firstNPrimes Frame	name:n, value:2
	name:nMinusOnePrimes, value:@1
	name:maxPrime, value:2

Heap

RAM @1
Object of type List[Int]
2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- The previous recursive call gets this returned reference
- Accesses that List on the heap

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
firstNPrimes Frame	name:n, value:2
	name:nMinusOnePrimes, value:@1
	name:maxPrime, value:2
findPrime Frame	name:i, value:3
	name:knownPrimes, value:@1

Heap

RAM @1
Object of type List[Int]
2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- The reference is passed to the next method call
- This is the reference behavior we expect

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
firstNPrimes Frame	name:n, value:2
	name:nMinusOnePrimes, value:@1
	name:maxPrime, value:2
findPrime Frame	returns 3

Heap

RAM @1
Object of type List[Int]
2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- Since 3 is not divisible by 2
  - Return the base case of i

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
firstNPrimes Frame	name:n, value:2
	name:nMinusOnePrimes, value:@1
	name:maxPrime, value:2
	getting return value of 3

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- Get return value of 3 and prepend it to the List of know primes
- But Lists are immutable!
  - Create a new List with 3 prepended

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
firstNPrimes Frame	returns the reference @2

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- A reference to the new List is returned
- The original List remains on the heap and is unchanged

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
firstNPrimes Frame	returns the reference @2

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- **Important**:
  - If another part of our program has the reference @1 stored in a variable
  - Nothing we do can interfere with its computation

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
	name:nMinusOnePrimes, value:@2
	name:maxPrime, value:3

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- The first recursive call gets the reference @2
- Continues its computation with this reference

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
	name:nMinusOnePrimes, value:@2
	name:maxPrime, value:3
findPrime Frame	name:i, value:4
	name:knownPrimes, value:@2

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- Make a call to findPrime based on the List @2
- Base case is false since 4%2 == 0

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
	name:nMinusOnePrimes, value:@2
	name:maxPrime, value:3
findPrime Frame	name:i, value:4
	name:knownPrimes, value:@2
findPrime Frame	name:i, value:5
	name:knownPrimes, value:@2

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

```scala
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- Recursive call is made to check if the next integer is prime

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
	name:nMinusOnePrimes, value:@2
	name:maxPrime, value:3
findPrime Frame	name:i, value:4
	name:knownPrimes, value:@2
findPrime Frame	returns 5

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- Hit the base case since 5 is prime
- Return 5 up the recursion

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
	name:nMinusOnePrimes, value:@2
	name:maxPrime, value:3
findPrime Frame	returns 5

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- Return 5 up the recursion

Stack

RAM	
Main Frame	args
firstNPrimes Frame	name:n, value:3
	name:nMinusOnePrimes, value:@2
	name:maxPrime, value:3
	getting return value of 3

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

RAM @3
Object of type List[Int]
5, 3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- With the return value of 5
  - firstNPrimes can finish its computation
- Create another new List on the heap

Stack

RAM	
Main Frame	args
firstNPrimes Frame	returns the reference @3

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

RAM @3
Object of type List[Int]
5, 3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- With the return value of 5
  - firstNPrimes can finish its computation
- Create another new List on the heap

Stack

RAM	
Main Frame	args
	gets return value of @3

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

RAM @3
Object of type List[Int]
5, 3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- Main gets the List at reference @3
- The other two Lists are still on the Heap and are unchanged

Stack

RAM	
Main Frame	args
	gets return value of @3

Heap

RAM @1
Object of type List[Int]
2

RAM @2
Object of type List[Int]
3, 2

RAM @3
Object of type List[Int]
5, 3, 2

```
def firstNPrimes(n: Int): List[Int] = {
  if (n < 1) {
    List()
  } else if (n == 1) {
    List(2)
  } else {
    val nMinusOnePrimes: List[Int] = firstNPrimes(n - 1)
    val maxPrime: Int = nMinusOnePrimes.max
    findPrime(maxPrime + 1, nMinusOnePrimes) :: nMinusOnePrimes
  }
}

def findPrime(i: Int, knownPrimes: List[Int]): Int = {
  if (!knownPrimes.foldLeft(false)(_ || i % _ == 0)) {
    i
  } else {
    findPrime(i + 1, knownPrimes)
  }
}


def main(args: Array[String]): Unit = {
  firstNPrimes(3).foreach(println)
}
```

- The primes 5, 3, and 2 are printed to the console

Lecture Question

Restriction: No state is allowed in this question. Specifically, the keyword "var" is banned

Question: In a package named "functions" write a class named Point with the following features:

- Has a constructor that takes 2 values (Use val) of type Double named "x" and "y"
- A method named "add" that takes a Point and returns a Point that is the component-wise addition of this Point and the input Point
  - Ex. (1.0, 2.0) + (4.0, 1.0) = (5.0, 3.0)
- A method named "multiplyByScalar" that takes a Double and returns a new Point that is this Point multiplied by the input
  - Ex. 5.0 * (1.0, 2.0) = (5.0, 10.0)

Testing: In a package named "tests" create a class named "TestPoint" as a test suite that tests all the functionality listed above